Analyse de performances du code ComPASS

Yacine Ould Rouis, Michel Kern, Simon Lopez

Projet ANR CHARMS*
Analyse de performances du code ComPASS

Yacine Ould Rouis†, Michel Kern‡, Simon Lopez§

Rédaction initiale août 2018, mise à jour le 24 août 2021

Table des matières

*Projet ANR-16-CE06-0009
†Maison de la Simulation, CEA – CNRS – Université Paris- Saclay
‡Inria, Centre de Paris, Paris
§BRGM, Orléans

0 Introduction

Dans ce rapport, nous exposons les travaux d'analyse de performance et d'optimisation réalisés sur le code ComPASS, dans le cadre de la participation de la Maison de la Simulation au projet CHARMS. Ces travaux ont eu lieu entre janvier et août 2018.

L'analyse de performance en HPC consiste à mettre en oeuvre un certain nombre de connaissances et outils afin d'obtenir une compréhension approfondie du comportement du code sur une plate-forme de calcul. Elle permet d'identifier, entre autres, les régions critiques du code, les goulots d'étranglement, et les configurations optimales d'utilisation. Elle s'inscrit dans un processus itératif qui a pour but l'amélioration générale du code et l'optimisation de ses performances en termes de temps de calcul et d'utilisation des ressources.

Les performances d'un code parallèle comportent plusieurs aspects :

Les performances parallèles englobent la scalabilité en fonction de la taille du problème et du nombre de rangs MPI, l'équilibrage de charge entre les rangs MPI, le temps passé dans les communications.

Les performances au niveau du nœud concernent l'utilisation des caches et de la mémoire, les comportements « compute-bound » (limitations par la vitesse de calcul du CPU) ou « memory-bound » (limitations par la bande passante des accès mémoire) des différentes parties du calcul, et la bonne exploitation des fonctionnalités qu'offrent les processeurs en termes de vectorisation et optimisations diverses.

Les performances numériques et algorithmiques On peut également améliorer le comportement du code indépendamment des considérations d'architecture, en analysant l'algorithme, son implémentation, son comportement, et en repérant, par exemple, des calculs redondants ou inutiles.

Les performances des entrées et sorties (E/S) focalisent sur les accès aux disques–durs.

Après une description du code ComPASS, et des cas d'utilisation nous présenterons les résultats de l'étude de performance. Une attention particulière est portée à deux aspects :

performances parallèles comment se comporte le code lorsque le nombre d'unités de calcul augmente ;

performances numériques Comment choisir au mieux les paramètres gouvernant la résolution du problème discret à chaque pas de temps, en insistant sur le contrôle des processus itératifs.

1 Présentation du code ComPASS

ComPASS [9, 3] simule les transferts de masse et d'énergie souterrains en milieux poreux fracturés avec prise en compte des effets thermiques. Il a des applications dans la modélisation de réservoirs et systèmes géothermiques de haute température.

1.1 Le modèle

Le modèle physique et sa réalisation numérique sont décrits en détail dans les références [9, 13]. Ils reposent sur la simulation de l'écoulement multiphasiques compositionnels

de fluides darcéens non-isothermiques sur un maillage 3D non structuré, dont les éléments peuvent être des polyèdres quelconques. Le modèle 3D est couplé à un réseau de fractures discrètes.

Le problème est discrétisé à l'aide d'une intégration implicite en temps combinée à un schéma de volumes finis *Vertex Approximate Gradient* (VAG) [4]. La formulation du modèle compositionnel choisie est de type Coats [4]. Elle permet de prendre en compte l'apparition ou la disparition d'une phase via un changement du système d'inconnues au cours des itérations à partir des considérations thermodynamiques. Le modèle prend en compte un nombre arbitraire de composants dans chaque phase permettant de modéliser les écoulements non miscibles, partiellement miscibles ou entièrement miscibles. Plusieurs équations d'états peuvent être implémentées.

Le problème non linéaire est résolu à chaque pas de temps à l'aide d'une méthode de type Newton. Chaque itération de Newton nécessite la résolution d'un système linéaire. Celui-ci est résolu par une méthode itérative de Krylov (en général GMRES [11]) préconditionnée. Le mauvais conditionnement de systèmes rencontrés rend nécessaire le choix d'une méthode de préconditionnement robuste. La méthode CPR-AMG [10, 8, 12], qui combine un multigril le algébrique pour le bloc de pression avec une méthode de Jacobi par bloc pour la matrice complète, a fait ses preuves dans des applications similaires.

La convergence de la méthode de Newton est utilisée pour adapter le pas de temps : celui-ci est augmenté (d'un facteur 1.2) si les itérations ont convergé dans le nombre d'itérations demandé, et réduit (de moitié) en cas d'échec de la convergence.

1.2 Le code

Le code est développé en Fortran 2003 pour le coeur de calcul, avec une structure en Python qui permet de lire et écrire les données de simulation, et de piloter la boucle en temps. Les deux parties sont liées par une interface C++, pybind11.

Le code est parallélisé à l'aide de MPI. Les systèmes entièrement couplés sont assemblés et résolus en parallèle avec une couche de cellules fantômes. Cette stratégie permet un assemblage local des systèmes discret et minimise la quantité de communications entre les rangs MPI.

Le solveur non linéaire de type Newton est implémenté en Fortran.

La résolution du problème linéaire fait appel à la bibliothèque PETSc [1, 2], qui permet un vaste choix de méthodes de résolution et de méthodes de préconditionnement sur des architectures parallèles. Le multigrille algébrique est l'algorithme BoomerAMG provenant de la biblitohèque Hypre [5]).

En sortie, les résultats du calcul sont stockés dans un fichier par processus MPI et par pas de temps d'écriture, à l'aide de la routine `numpy.savez`. Ils peuvent ensuite être convertis en un format lisible par les logiciels de visualisation, tels que Parvaiew.

Le code est sous licence CeCILL v2.1 / GNU GPL V3

2 Les cas d'étude

Le choix d'un cas représentatif est l'un des points les plus déterminants de l'analyse de performance, et repose sur une bonne communication et la compréhension des besoins des porteurs et utilisateurs du code. Un bon choix réside dans le fait que le cas doit être à la fois

— économe en temps et en ressources, afin d'avoir un temps de restitution et une consommation d'heures de calcul raisonnables

— le plus proche possible d'un cas de production, en terme de nature et quantité de calculs et d'utilisation du matériel, afin d'obtenir des résultats d'analyse pertinents.

2.1 Le cas doublet (monophasique)

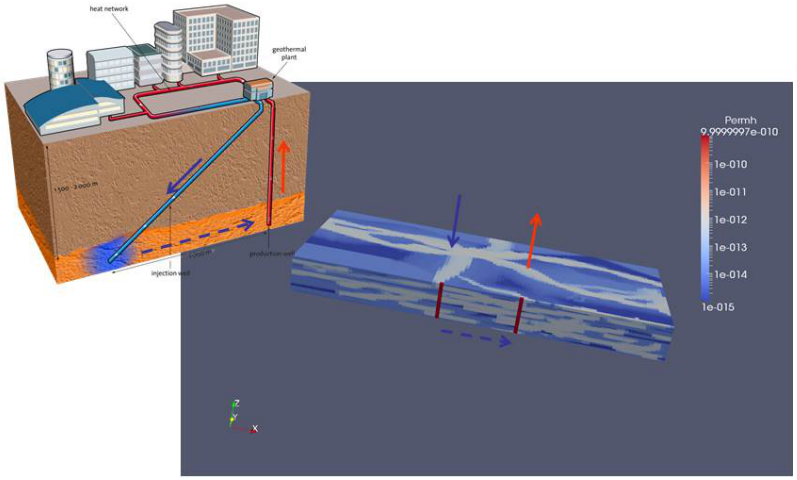

FIGURE 1 – Typique doublet géothermique qui consiste en une boucle fermée avec un puits de production chaud (rouge) et un puits d'injection froid (bleu). L'injection d'eau froide conduit à l'épuisement progressif et temporaire de la ressource à l'échelle locale du doublet.

Le cas *doublet* sur grille cartésienne, un premier cas simple, a été fourni dans un premier temps. Il consiste en l'injection d'eau froide dans un réservoir d'eau chaude, tel qu'illustré dans la figure 1. Le modèle contient une seule phase, l'eau à l'état liquide, qui n'est pas sujette à l'évaporation, donc pas d'apparition d'une phase gazeuse au cours de la simulation.

Le modèle comporte une géométrie simple, et un maillage structuré. Il comporte environ 98000 inconnues.

Il a permis d'obtenir les premiers résultats, et une première idée des performances.

2.2 Le cas diphasique

Un deuxième cas plus complexe a été choisi par la suite. Il s'agit d'un modèle diphasique décrit dans [13], avec une possible apparition de phase et la prise en compte d'un réseau de fractures, sur un maillage non structuré.

Plusieurs maillages de tailles différentes sont fournis, ce qui permet une meilleure compréhension de la scalabilité. Les différents maillages sont de tailles : 450K, 800K, 1500K, 3M et 6M éléments. Ces différentes tailles de maillage se traduisent en systèmes linéaires dont le nombre d'inconnues est indiqué dans le tableau 1

Le modèle décrit précédemment sera appelé dans la suite *diphasique (1)*. Il se décline en une variante appelée cas *diphasique (2)*, avec des différences au niveau des perméabilités et des conditions aux bords. Ce second cas permet l'apparition de gaz dès le début de la simulation, ce qui provoque des difficultés pour le solveur non linéaire, qui va diverger et contraindre la réduction du pas de temps. Ce phénomène n'apparaît que plus tard dans le cas initial, et son étude aurait nécessité beaucoup plus de ressources sur les moyens de calcul disponibles.

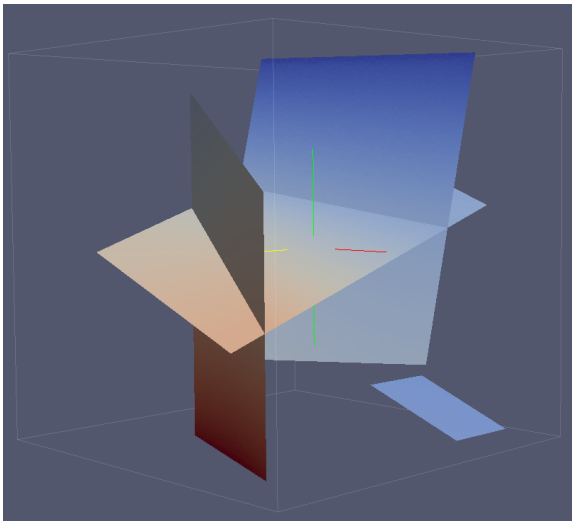

FIGURE 2 – Illustration du réseau de fractures dans le cas *diphasique* test-Xing17

Taille (nombre de cellules)	nombre d'inconnues
450K	1.6×10^5
800K	2.9×10^5
1500K	5.5×10^5
3M	1.1×10^6
6M	2.2×10^6

TABLE 1 – Nombre estimé d'inconnues pour chaque taille de maillage, cas *diphasique*

3 La mesure des performances

L'analyse de performance dont les résultats sont résumés dans ce document met en oeuvre un certain nombre d'outils et de méthodes qui sont brièvement présentées dans cette section.

3.1 Outils

JUBE[1] est un environnement de benchmarking qui permet d'automatiser les routines de test. Il permet de réaliser des tests multiparamètriques robustes et reproductibles en en maîtrisant toutes les étapes : de la récupération des sources, la paramétrisation, la définition de l'environnement, la compilation, l'exécution, à la récupération de résultats clés. L'interface de l'outil se base sur un script XML pouvant intégrer des blocs en Python.

Score-P and Scalasca[2],[3] La suite constituée par Score-P [7] et Scalasca [6] permet de collecter et visualiser des mesures de performance sous forme de profils et traces d'exécution d'un code parallèle, permettant d'identifier les temps passés dans chaque routine, et au niveau de chaque rang MPI ou thread OpenMP. Ces outils se basent sur l'instrumentation automatique du code à la compilation. Ils sont pertinents pour étudier le comportement

1. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/JUBE2/_node.html`
2. `https://www.vi-hps.org/projects/score-p`
3. `http://scalasca.org`

parallèle du code et identifier les déséquilibres de charge et la raison de ces déséquilibres. Cependant, l'instrumentation peut introduire un sur-coût sur les temps d'exécution et l'utilisation de la mémoire, et ainsi fausser les résultats. Parfois même une utilisation experte ne suffit pas à réduire ce sur-coût.

VTune, Advisor [4] Ces outils permettent un profiling plus fin du code, allant jusqu'au niveau des boucles, des instructions, voire des instructions assembleur. Cela en fait de très bons outils pour analyser les performances au niveau d'un nœud, identifier les instructions les plus coûteuses, optimiser les boucles et améliorer la vectorisation. Ces outils se basent principalement sur l'échantillonnage, ce qui, sauf cas pathologique, ne produit pas de surcoût.

Darshan [5] permet d'analyser et tracer les accès aux disques et fourni plusieurs informations concernant les temps, les volumes et la nature des appels des entrées – sorties.

3.2 Méthodes

L'instrumentation manuelle : Bien que limité et intrusif, cela reste le moyen le plus fiable et précis pour surveiller les temps passés dans les différentes parties du code.

L'étude de scalabilité : La scalabilité forte (taille du problème constante, on augmente n, le nombre de CPUs), et la scalabilité faible (taille du problème augmente proportionnellement à n) sont deux indicateurs différents de l'efficacité de la parallélisation du code.

La scalabilité sur n rangs (resp. nœuds) par rapport à 1 rang (resp. nœud) peut s'exprimer par deux indicateurs différents :

Le speedup (ou facteur d'accélération) $S(n) = \dfrac{T(1)}{T(n)}$, où $T(p)$ est le temps écoulé pour l'exécution du programme sur p processeurs ;

L'efficacité $E(n) = \dfrac{T(1)}{nT(n)} = \dfrac{S(n)}{n}$, qui est par définition entre 0 et 1.

L'étude mémoire vs CPU : Le test de limitations « mémoire vs cpu » consiste à comparer les temps d'exécution, à nombre de rangs MPI égal, sous deux configurations distinctes, le mode « regroupé » et le mode « éparpillé ». Dans les deux cas, on utilise la moitié des cœurs disponibles sur un nœud de calcul, qui est généralement composé de deux sockets.

— Dans le mode « regroupé », on utilise tous les cœurs d'un socket et le deuxième socket est vide.

— Dans le mode « éparpillé », on répartit les rangs équitablement entre les deux sockets.

La différence des temps d'exécution entre les deux modes nous renseigne sur l'effet de la bande passante mémoire sur les performances. Ainsi, si le mode « regroupé » est sensiblement plus lent, cela dénote d'une bande passante saturée, et donc d'un comportement limité par les accès mémoire (ou « memory bound »). Si le mode « regroupé » a un temps d'exécution proche du mode »éparpillé », la bande passante mémoire n'est pas saturée, et le code est donc limité par la vitesse de calcul du CPU (ou « CPU bound »).

L'indice qui sera affiché en résultat de ce test dans la suite de ce document est le sur-coût, en %, du cas « regroupé » par rapport au cas « éparpillé ».

4. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
5. `https://www.mcs.anl.gov/research/projects/darshan/`

Vectorisation et FMA Afin d'évaluer l'efficacité de l'utilisation des fonctionnalités offertes par les processeurs en termes de vectorisation et de l'utilisation du « FMA » (en anglais « fuse-multiply-add », c'est à dire la fusion d'une addition et d'une multiplication en une opération) , il est possible de désactiver ces deux fonctionnalités à la compilation, et ainsi avoir une base de comparaison.

L'indice d'efficacité qui sera affiché en résultat de ces tests dans la suite du document est l'accélération, en pourcentage, lorsqu'on active l'option, par rapport à l'option désactivée.

3.3 Mise en place des tests

Avant de présenter les résultats de manière synthétique, il est intéressant de dire quelques mots sur les hypothèses initiales, la préparation du code et la plate-forme de benchmarking automatisée, et les premiers tests qui ont permis de converger vers une analyse pertinente.

Une analyse dans le cadre du projet H2020 EoCoE a permis de réaliser un premier script JUBE intégrant la compilation et l'exécution automatiques du code avec différents outils de mesure de performances, et différents modes de compilation.

Ce travail et ses résultats ont constitué une base pour l'analyse que nous sommes sur le point de présenter. Le script JUBE a d'abord été adapté à la nouvelle version du code ComPASS qui est passé entre temps d'un code pur Fortran à un code hybride Python/Fortran/C++, et les nouvelles versions des différents outils de mesure de performances mis en place.

De nouvelles fonctionnalités ont ensuite été ajoutées, pour permettre de contrôler différents paramètres au niveau du code, des solveurs, de la compilation, du cas d'usage et de l'exécution. Cela comprend entre autres :

- le choix de la branche git ou des sources à copier ;
- dans les sources, le contrôle des paramètres des solveurs linéaire et non linéaire (nombre d'itérations max, tolérance, ...) ;
- le contrôle des options de compilation, les options d'optimisation, le choix du compilateur ;
- l'intégration des outils d'analyse de performance VTune et Advisor [6] ;
- le choix des cas test, de la taille du maillage, du temps de simulation final, et les intervalles des E/S ;
- la configuration des scripts d'exécution : le nombre de noeuds et de rangs MPI, la queue d'exécution, ... ;
- la possibilité de lancer simultanément plusieurs cas de tailles différentes sur plusieurs configurations différentes, pour des études de scalabilité par exemple.

Quelques instructions ont également été ajoutées à différents endroits dans le code ComPASS afin de mesurer sans sur-coût les temps d'exécution des différentes parties et différents calculs. Ces modifications n'ont pour l'instant pas été reportées dans la branche principale du code.

Le super-calculateur utilisé pour les tests est la machine Jureca (*Jülich Research on Exascale Cluster Architectures*) située au Centre de Recherche de Jülich en Allemagne, dans sa configuration d'avril 2017. La machine compte 1872 noeuds, chaque noeud comporte deux processeurs, avec chacun 12 coeurs Intel E5-2680 v3 Haswell CPUs à 2.5GHz, et chaque noeud possède 128 Go de mémoire.

6. Un problème lié à l'absence de la table des symboles à la compilation en -g a été résolu grâce à l'option CMake `-D CMAKE_STRIP=OFF`

4 Performances générales

Cette partie aborde les résultats généraux, et les premiers profils d'exécution, sur 2 noeuds de calcul, soit 48 cœurs physiques, pour deux cas choisis : le cas *doublet*, et le cas *diphasique* 3M.

	Métrique	cas doublet	cas diphasique 3M
Global	Temps Total (s)	48	4196
	Temps E/S (s)	9.65	1.98
	Temps MPI	18s (35%)	860s (20%)
	Mémoire vs CPU	10%	23%
	Déséquilibre de charge	28%	entre 8 et 15%

TABLE 2 – Résultats globaux de l'analyse de performances pour le cas *doublet* (98k inconnues) et le cas *diphasique* 3M (1.1M inconnues) sur 2 nœuds, soit 48 cœurs

4.1 Le cas doublet

Le temps d'exécution du cas *doublet* varie beaucoup d'une exécution à l'autre, entre 40 et 53 secondes (le temps rapporté dans la Table 2 correspond à une valeur moyenne). Cette variation, comme on le verra au chapitre 7, est due aux variations des temps d'accès aux disques, ou temps d'entrée/sortie en d'autres termes, qui occupent dans le cas présent près de 10 secondes.

Le temps des communications est très significatif, occupant 35% du temps d'exécution, dont 28% sont dus aux déséquilibres de charge, causant des temps d'attente lors des étapes de synchronisation. Ces déséquilibres sont dûs à l'étape d'initialisation, lors de laquelle seul le processus 0 génère et partitionne le maillage régulier, au niveau des entrées/sorties, mais aussi au niveau du solveur.

L'étude « mémoire vs cpu » décrite au paragraphe 3.2 montre un comportement légèrement limité par la bande passante mémoire.

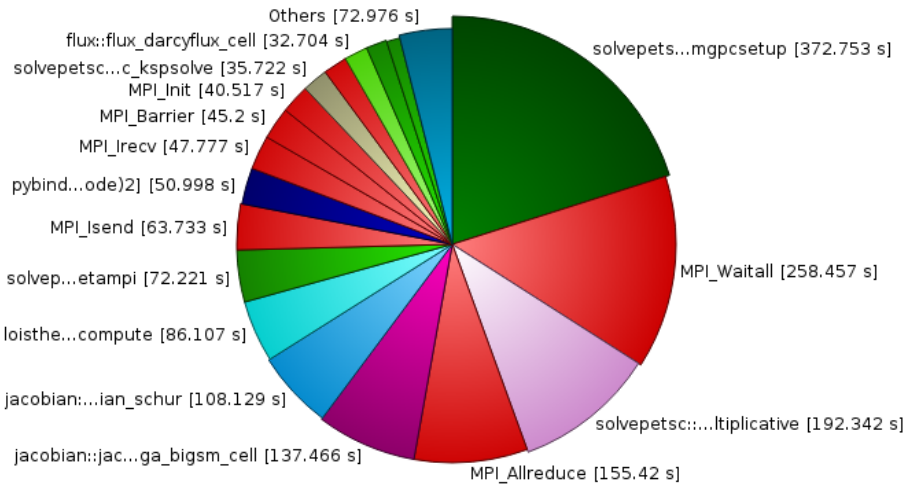

FIGURE 3 – Profil Scalasca/Vampir du cas *doublet* sans écriture des résultats

L'étude Scalasca permet d'obtenir le profil détaillé dans la figure 3. L'écriture des résultats est désactivée dans ce cas, le temps des E/S devient alors négligeable et le temps

d'exécution est stable, égal à 39 secondes. Sur la figure, les temps affichés sont les temps CPU, autrement dit le cumul des temps pour chacun des 48 processus MPI, ce qui explique les valeurs bien plus importantes.

Sans surprise, l'on y voit un temps d'exécution dominé par les temps de communications MPI, en rouge. Les communications individuelles asynchrones, que représentent `MPI_Isend`, `MPI_Irecv` et `MPI_Waitall`, ou collectives (`MPI_barrier` et `MPI_AllReduce`), y sont prédominantes et représentent un peu plus de 30% du temps total. Ces fonctions sont appelées principalement par le préconditionneur et le solveur linéaires, qui sont des routines de la librairie PETSc, mais également par une fonction de synchronisation des points « fantômes » à la fin de chaque itération de Newton. Les temps mesurés des autres routines n'incluent pas les temps MPI. En tête de ceux-ci, on retrouve le préconditionnement du problème linéaire (`solvepetsc_cpramgsetup`, plus de 20%), suivi du solveur linéaire (`solvepetsc_cpramgapply_p_multiplicative environ 10%`) et des deux routines composant le calcul de la jacobienne (`jacobian_compute_biga_bigsm_cell` et `jacobian_schur`), pour un total de 15%.

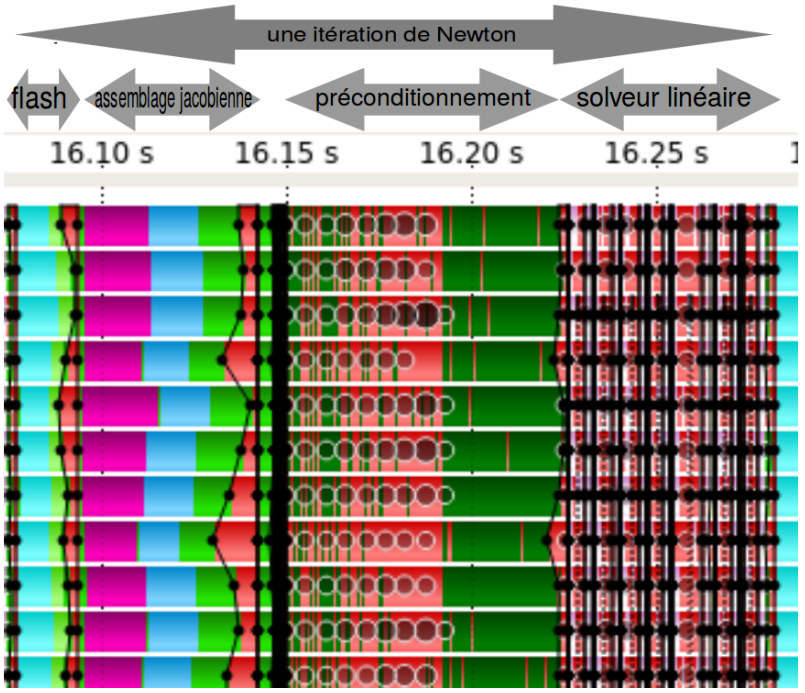

FIGURE 4 – Trace Scalasca/Vampir représentant une itération de Newton du cas *doublet*, suivant le même code couleur que la figure 3

La figure 4 représente un extrait de la trace d'exécution, au cours d'une itération de Newton. L'axe horizontal est le temps, et chaque ligne représente un processus MPI. Cette représentation donne une idée plus détaillée du comportement parallèle et des interactions entre les processus, et permet une analyse visuelle des déséquilibres. On peut y identifier les trois parties principales décrites plus haut, et y remarquer le grand nombre de communications appelées depuis le solveur et son préconditionneur.

Il apparaît au bout de cette étude que ce cas n'est pas représentatif d'une bonne utilisation du code, du solveur en termes de taille par rapport à la configuration choisie, et de fréquence des E/S. Les résultats obtenus donnent toutefois une bonne idée du fonctionnement du code, et permettent déjà d'en repérer les points critiques.

4.2 Le cas diphasique

Le cas *diphasique* 3M sur 50 années simulées et sur 48 cœurs de calcul s'exécute en 70 minutes. Les E/S affichent un bien meilleur débit, et occupent un temps négligeable. Une analyse plus détaillée en est fournie au chapitre 7. Le temps de communication est moins important dans ce cas d'utilisation, occupant 20% du temps total. Ce temps MPI se retrouve toujours principalement au niveau des solveurs et préconditionneurs PETSc.

Il n'a pas été possible d'obtenir des mesures Scalasca exploitables, essentiellement à cause du grand nombre d'appels MPI qui surcharge la mémoire lors de la collecte de la trace, il est donc difficile d'obtenir une estimation du taux de déséquilibre du code. Celui-ci semble se situer entre 8 et 15%.

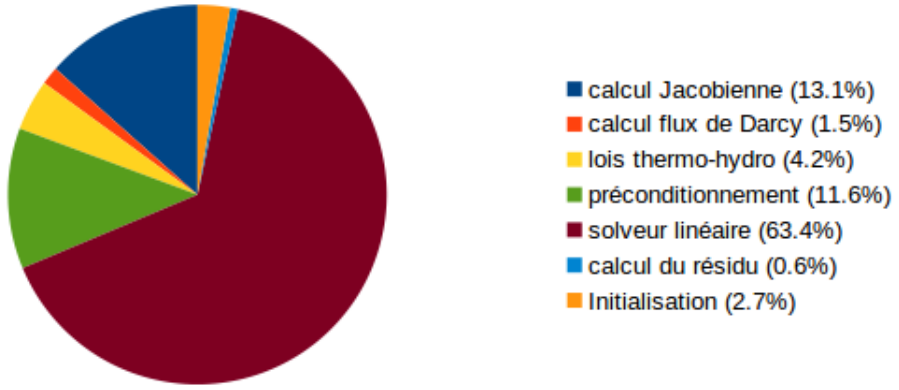

FIGURE 5 – Profil d'exécution du cas *diphasique* sur 48 cœurs, par instrumentation manuelle

Le profil d'exécution est obtenu, cette fois, à l'aide d'une instrumentation manuelle des différentes parties d'intérêt. Les communications ne sont pas mesurées indépendamment des régions du code qui les appellent. Le résultat, présenté sur la figure 5, montre un temps d'exécution dominé par le solveur linéaire qui en occupe près des deux tiers. Il est suivi par l'assemblage de la matrice (Calcul Jacobienne) et le préconditionnement. Ce comportement, conforme à ce qui est attendu dans ce genre de problème (la complexité du solveur linéaire est supérieure à celle de l'assemblage, et doit donc dominer pour des problèmes de grande taille) est plus représentatif que le cas précédent.

En ce qui concerne les performances au niveau du nœud, l'étude « mémoire vs cpu » permet d'identifier un comportement plus « memory-bound », c'est à dire limité par la bande passante mémoire. Cela peut s'expliquer par plus d'accès mémoire, car les données dans ce cas sont de multiples fois plus grandes que dans le cas précédent. Le maillage également, n'étant plus régulier, peut être la raison de plus d'indirections des accès mémoire. Le résultat détaillé de ce test est affiché dans le tableau 3. Le solveur est la partie la plus impactée par la bande passante mémoire, avec un ralentissement de 37%.

	calcul total	jacobienne	flux Darcy	lois t-h	précond.	solveur
6 rangs par socket	3190 s	501 s	57 s	169 s	458 s	1939 s
12 rangs par socket	4009 s	552 s	61 s	172 s	487 s	2655 s
Ralentissement	26%	10%	7%	1.5%	6%	37%

TABLE 3 – Le test « mémoire vs cpu » détaillé pour les différentes régions mesurées, pour le cas *diphasique* 3M sur 48 cœurs.

Enfin, l'analyse de la vectorisation et le FMA ne montre pas de gains dûs à ces fonctionnalités. Cette analyse ne concerne que les parties du code compilées au moment de l'installation de ComPASS, et ne prend pas en compte les routines des librairies externes, telles que le préconditionnement et le solveur, puisque PETSc est en général déjà installé sur les super-calculateur.

5 Scalabilité

Une large étude de scalabilité a été réalisée pour le cas *diphasique*. Nous exposons ici les résultats de scalabilité intranodale, c'est à dire de 1 à 24 cœurs au niveau d'un nœud de calcul, et la scalabilité internodale, de 1 à 16 nœud de calcul (de 24 à 384 cœurs).

Cette étude permet d'étudier le passage à l'échelle du code sur un nombre croissant de processus MPI, et d'identifier par la même occasion la configuration idéale (en termes de nombre de processus et utilisation du moyen de calcul) pour une taille de problème donnée.

Les mesures ont été effectuées globalement pour le temps de calcul total, et en détail pour chaque partie d'intérêt dans le code à l'aide de l'instrumentation manuelle. Les tests ont tous été réalisés sur JURECA.

5.1 Scalabilité intranodale

La scalabilité intranodale a été mesurée pour le cas 1500K, qui conduit à un système linéaire avec environ 500 000 inconnues, sur des configurations de 1 à 24 cœurs, équitablement répartis sur les deux sockets du nœud de calcul. L'efficacité est mesurée par rapport à la performance sur 1 cœur. Le choix du cas est dû à un soucis d'économie des heures de calcul. Le résultat est présenté sur la figure 6.

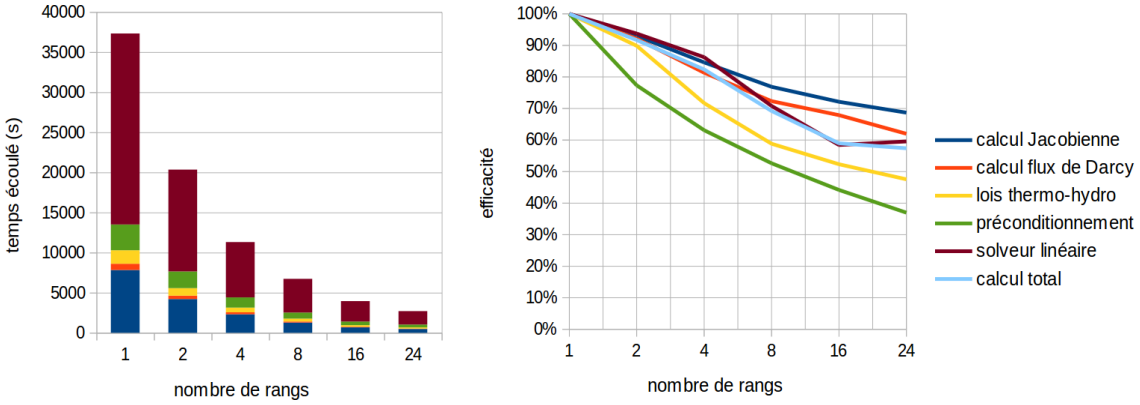

FIGURE 6 – Scalabilité forte intranodale détaillée pour le cas *diphasique* 1500K

Pour la taille du problème considéré, le nombre d'inconnues par coeur devient inférieur à 100 000 à partir de 8 coeurs, et tombe à 20 000 pour 24 coeurs. Dans ces conditions, les routines qui ne comportent pas de communications (celles qui contribuent au calcul de la Jacobienne) présentent une scalabilité qui reste supérieure à 70%, mais la préparation du préconditionnement (rappelons qu'il s'agit d'une méthode multi-grille algébrique) tombe rapidement en dessous de 60%, et ne dépasse pas 40% sur le noeud complet. Le calcul total est à 70% d'efficacité pour 8 coeurs, et autour de 60% pour 24 coeurs, ce qui est acceptable.

5.2 Scalabilité internodale

L'étude de scalabilité internodale a été réalisée pour chaque taille du cas *diphasique*, de 450K à 6M éléments, sur des configurations allant de 1 à 16 nœuds de calcul, c'est à dire de 24 à 384 cœurs.

Le même résultat est présenté ici dans la figure 7 sous 2 formes différentes : les deux courbes de gauche représentent les temps d'exécution, et la courbe de droite représente l'efficacité de la scalabilité par rapport aux temps sur 1 nœud (24 cœurs). Pour des raisons de lisibilité, les temps correspondant aux deux plus grosses configurations (3 millions et 6 millions de tétraèdres respectivement) sont sur une figure séparée des trois plus petites.

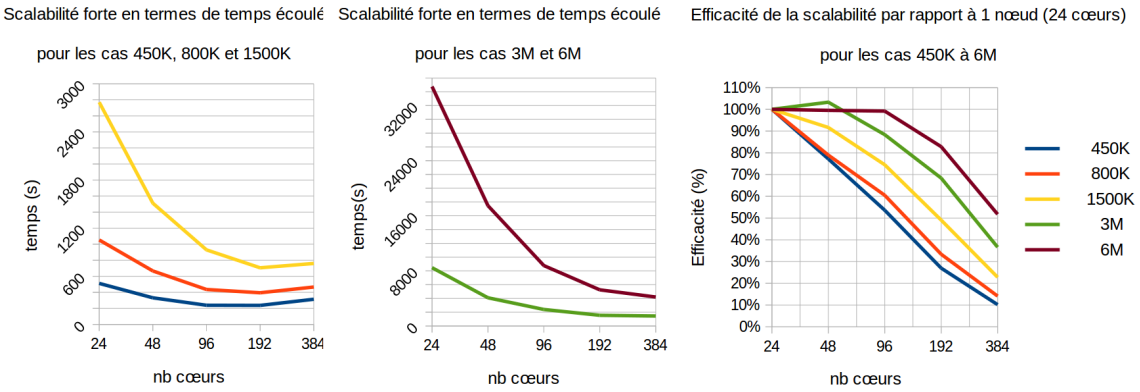

FIGURE 7 – Scalabilité du cas *diphasique*, pour différentes tailles de maillage

L'on remarque que, comme prévu, l'efficacité de la scalabilité augmente en fonction de la taille du problème. Ainsi les résultats pour les plus petits cas, en dessous de 1500K donnent des scalabilité assez faibles (en dessous de 50% dès que l'on dépasse les quatre noeuds). Le cas 3M a une scalabilité correcte jusqu'à 96 cœurs, et la scalabilité pour le cas 6M est excellente jusqu'à 96 cœurs, correcte jusqu'à 192 cœurs, puis se détériore fortement au delà.

Les écarts de performance entre les différentes tailles de problème laissent penser que les performances dépendent aussi des particularités des maillages.

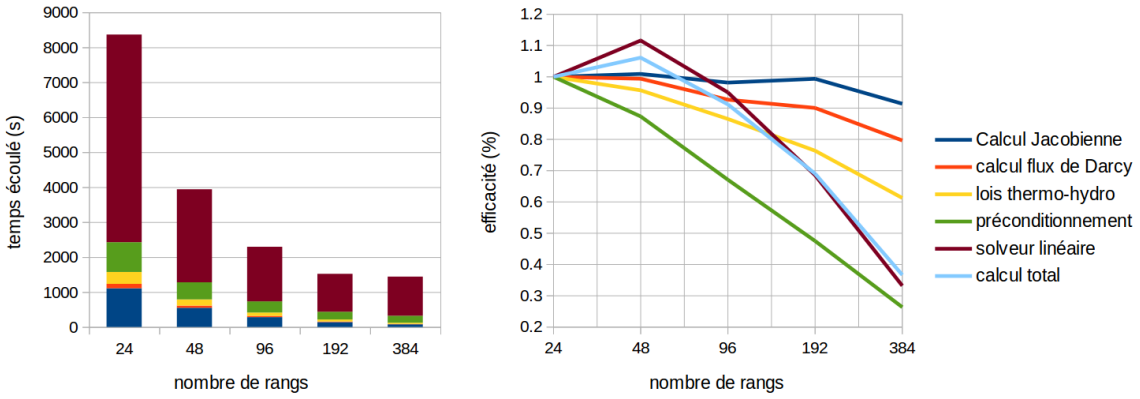

FIGURE 8 – Scalabilité forte internodale détaillée pour le cas *diphasique* 3M

Comme le montrent les résultats détaillés pour le cas 3M, figure 8, la scalabilité du calcul total est très dépendante de la scalabilité du solveur. Celui-ci trouve un régime idéal sur 48 cœurs, produisant une hyperscalabilité sur cette configuration, puis affiche une

scalabilité supérieure à 90% sur 96 cœurs, avant de perdre fortement en efficacité au delà. Des mesures faites par ailleurs (voir le paragraphe 4), montrent que la perte d'efficacité est accompagnée par une augmentation des temps de communication MPI.

Cela est cohérents avec les recommandations PETSc [7], qui prévoient une détérioration des performances du solveur KSP en dessous d'un certain nombre d'inconnues par rang. Il faut en effet une charge de travail assez conséquente pour chaque processus afin que le temps de calcul l'emporte sur les temps de communication. PETSc recommande un minimum absolu de 10 000 inconnues par rang, et de préférence 20 000 inconnues ou plus.

Dans notre cas d'utilisation, le cas 3M génère un système linéaire qui comporte environ 1.1×10^6 inconnues. Le minimum recommandé est donc atteint sur 96 cœurs, avec environ 11500 inconnues par cœur, et franchi au delà.

Le préconditionnement, qui dépend également entièrement de la librairie externe PETSc, est le moins scalable des routines mesurées. Il est important de souligner que les systèmes linéaires résolus lors de ces simulations sont mal conditionnés, donc difficiles à résoudre (nous reviendrons sur ce point au paragraphe 6). En conséquence la méthode de préconditionnement mise en oeuvre (CPR-AMG [10, 12]) a été choisie pour sa robustesse (voir Table 5), même si sa scalabilité parallèle est plus faible que d'autres méthodes comme Jacobi par bloc. Cela explique en grande partie les efficacités relativement modestes qui sont observées.

Par contre, le calcul des jacobiennes est une partie parfaitement parallèle, ne faisant appel à aucune communication, grâce aux choix d'une décomposition avec recouvrement comprenant des cellules fantômes. Cela en fait une partie très bien scalable.

6 Performances numériques

Ce chapitre aborde les aspects numériques des tests présentés jusque là sur le cas *diphasique*. Cela inclut les comportements du solveur linéaire, appelé dans la suite KSP d'après la terminologie de PETSc, et du solveur non linéaire de Newton.

Comme expliqué dans le chapitre 2, le cas *diphasique* a été testé sous deux variantes qui diffèrent au niveau des propriétés physiques et des conditions au bord. Ces deux variantes ont un comportement similaire en termes de scalabilité et de proportions au niveau des profils d'exécution, mais elles diffèrent au niveau des performances numériques (et donc également au niveau des temps d'exécution), et il sera pertinent dans ce chapitre de les traiter séparément. Cela permettra de mettre en évidence des problèmes différents et suggérer des solutions efficaces.

La variante du cas *diphasique* étudié dans les chapitres précédents sera nommée *"cas diphasique (1)"*. Nous ferons référence à la deuxième variante sous le nom de *"cas diphasique (2)"*.

Les métriques étudiées ici sont :

7. PETSc FAQ `http://www.mcs.anl.gov/petsc/petsc-3.2/docs/faq.html#slowerparallel`

métrique	description
n_{pdt}	le nombre de pas de temps réussis
$n_{\text{échec Newton}}$	le nombre de pas de temps où le solveur non-linéaire n'a pas convergé
$n_{\text{iter Newton}}$	le nombre d'itérations du solveur non-linéaire de Newton, des pas de temps qui convergent [8]
$n_{\text{échec KSP}}$	Le nombre d'itérations du solveur non linéaire qui ont échoué à cause d'un défaut de convergence du solveur
$n_{\text{iter KSP}}$	Le nombre d'itérations du solveur linéaire KSP calculés
$n_{\text{iter KSP utile}}$	Le nombre d'itérations du solveur KSP, ne prenant pas en compte les pas de temps non convergés

Ces métriques varient en fonction de la taille du problème, il sera donc pertinent de les présenter pour chaque taille. Par contre elles ne varient que légèrement en fonction du nombre de rangs MPI, et de manière non corrélée à ce nombre, avec un aléa qui peut aller de 5% à 20%. Cela peut être dû aux différences introduites lors du partitionnement du problème. Les résultats seront donc présentés avec une approximation de 20%, indépendamment du nombre de rangs MPI.

6.1 Le cas diphasique (1)

Les résultats numériques de ce cas sont synthétisés dans le tableau 4.

cas	n_{pdt}	$n_{\text{échec Newton}}$	$n_{\text{iter Newton}}$	$n_{\text{échec KSP}}$	$n_{\text{iter KSP}}$
450K	85	0	646	0	24935
800K	85	0	640	0	30695
1500K	89	0	674	1	42221
3M	169	0	1010	26	77663
6M	444	0	1905	100	159056

TABLE 4 – Résultats de performances numériques, cas *diphasique (1)* sur 50 ans

Les nombre de pas de temps et d'itérations de Newton semblent stables et équivalents pour les plus petits cas, sans aucun échec des solveurs Newton et KSP. À partir de 1500K, nous observons une augmentation du nombre de pas de temps et d'itérations de Newton, accompagnée par l'apparition d'échecs de convergence au niveau du solveur linéaire.

L'échec de convergence d'un solveur se produit est lorsque le nombre d'itérations maximum, défini dans le code, est atteint. Cela cause l'annulation du pas de temps en cours de calcul, et la réduction du pas δt de moitié. Si le nombre de réductions du pas de temps dépasse une limite fixée par l'utilisateur, le calcul s'arrête. Au contraire, lorsque le calcul au niveau d'un pas de temps converge, le pas δt est doublé pour le pas de temps suivant jusqu'à un maximum défini par l'utilisateur.

Par défaut dans le code, la limite du nombre d'itérations pour les systèmes linéaires est fixée à 150 itérations. Le tableau 5 montre une augmentation du nombre moyen d'itérations du solveur linéaire en fonction de la taille du problème. Cette moyenne est toutefois faussée par une lente progression du nombre d'itérations au début de la simulation (qui coïncide

8. Cette métrique affichée par le code à la fin de la simulation est légèrement inexacte, car la boucle en temps incrémente une itération de Newton de trop par pas de temps. Le nombre exact devrait donc être le nombre affiché moins le nombre de pas de temps Cette correction est effectuée dans les résultats affichés plus loin.

avec l'augmentation progressive du pas de temps), comme on peut le voir, pour le cas 3M, en bleu sur la figure 9.

Cas	Iters. Newton	Iters KSP
450K	7.6	44
800K	7.5	55
1500K	7.6	71
3M	6.0	87
6M	4.3	96

TABLE 5 – Nombre d'itérations moyen pour la convergence des solveurs, cas *diphasique (1)* sur 50 ans

Pour les trois maillages de plus grande taille, nous pouvons augmenter la limite d'itérations maximum du solveur linéaire de 150 à 300. Les résultats correspondant sont résumés dans le tableau 6. Les échecs de convergence, avec `KSPitmax` = 300 sont quasiment éliminés, et le nombre de pas de temps et itérations de Newton devient quasiment équivalent aux petits cas. Les temps d'exécution des cas 3M et 6M sont réduits, respectivement, de 32% et 57%.

Dans des cas plus problématiques, si la limite doit être poussée plus loin, il peut être intéressant également de régler le paramètre « restart » de la méthode GMRES employée. Mais il faut être conscient qu'augmenter ce paramètre augmentera la mémoire nécessaire.

cas	n_{pdt}	$n_{\text{échec Newton}}$	$n_{\text{iter Newton}}$	$n_{\text{échec KSP}}$	$n_{\text{iter KSP}}$
1500K	85	0	651	0	39952
3M	86	1	661	0	55440
6M	88	0	701	1	73279

TABLE 6 – Résultats de performances numériques, cas *diphasique (1)* sur 50 ans, avec `KSPitmax`=300

Les figures 9 et 10 représentent l'évolution du nombre d'itérations KSP et Newton pour le cas 3M, à chaque appel de ces solveurs. On peut y identifier les échecs de convergence de KSP dans le cas `KSPitmax` = 150 qui rallongent l'exécution. On peut également identifier l'échec de convergence de Newton unique dans le cas `KSPitmax` = 300, avec une phase correspondante sous forme de « creux » dans la courbe des itérations KSP, où le solveur linéaire a un comportement différent différent du reste de la simulation.

6.2 Le cas diphasique (2)

Les résultats numériques de ce cas sont synthétisés dans le tableau 7.

Dans ce cas, qui est exécuté dans les mêmes conditions que le cas précédent, les solveurs convergent en moins d'itérations. Ce cas nous intéresse particulièrement pour les échecs répétés qui se produisent au niveau du solveur non linéaire. De telles phases de calcul peuvent survenir également dans le cas *diphasique (1)* mais plus loin dans la simulation, il est donc intéressant d'en étudier les performances.

Nous observons donc une dizaine d'échecs au cours des 50 années simulées, pour tous les cas, à l'exception du cas 3M, qui affiche un plus haut taux de défaut de convergence.

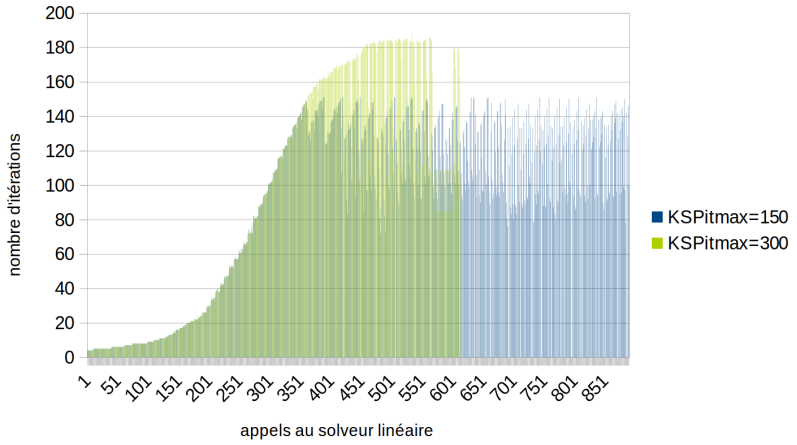

FIGURE 9 – Nombre d'itérations pour chaque appel du solveur linéaire KSP, avant et après la modification de KSPitmax. Pour le cas *diphasique (1)* 3M, sur 50 ans

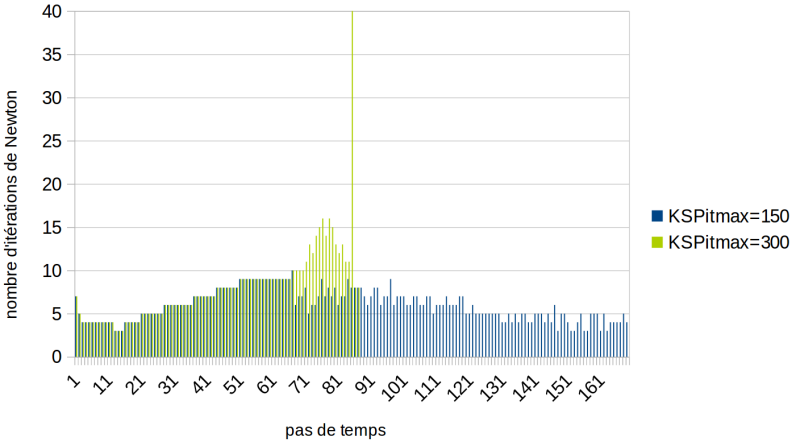

FIGURE 10 – Nombre d'itérations de Newton à chaque pas de temps, avant et après la modification de KSPitmax. Pour le cas *diphasique (1)* 3M, sur 50 ans

Ce nombre paraît faible par rapport au nombre de pas de temps convergeants. Cependant, lorsqu'il y a convergence, celle-ci est atteint en très peu d'itérations de Newton (cf. tableau 8), tandis qu'un échec n'est déclaré qu'au bout de `NewtonNiterMax` = 40 itérations de Newton.

Un simple calcul permet alors d'estimer la quantité de calcul que cela représente. Prenons le cas 450K, 9 échecs du Newton représentent 360 itérations de Newton calculées, soit près de 2.5 fois plus que l'ensemble des itérations calculées dans les pas de temps qui ont convergé. Cela se répercute sur le nombre total d'itérations KSP calculées $n_{\text{iter KSP}}$, en comparaison au nombre d'itérations $n_{\text{iter KSP utile}}$ affiché par le code à la fin de la simulation, qui ne prend en compte que les pas de temps convergeants. De plus, l'examen de l'historique de convergence sur quelques itérations où la méthode e Newton ne converge pas montre une stagnation des itérations qui se manifeste très tôt. Autrement dit, soit la méthode de Newton converge en un petit nombre d'itérations, soit elle ne converge pas du tout. La limite d'itérations du solveur non linéaire `NewtonNiterMax` = 40 semble donc trop grande et inadaptée au problème, pour ces phases de calcul où le solveur non linéaire ne converge pas toujours.

Des tests ont été réalisés en prenant de plus petites valeurs de `NewtonNiterMax`. Les

cas	n_{pdt}	$n_{\text{échec Newton}}$	$n_{\text{iter Newton}}$	$n_{\text{échec KSP}}$	$n_{\text{iter KSP}}$	$n_{\text{iter KSP utile}}$
450K	52	9	148	0	18708	4277
800K	59	12	161	0	23252	5551
1500K	51	8	154	0	20527	6447
3M	96	22	294	1	55391	13594
6M	56	10	284	0	45072	17954

TABLE 7 – Résultats de performances numériques, cas *diphasique (2)* sur 50 ans, `NewtonNiterMax` = 40

cas	Solveur de Newton	Solveur KSP
450K	2.8	29
800K	2.7	34
1500K	3.0	42
3M	3.1	46
6M	5.1	63

TABLE 8 – Nombre d'itérations moyen pour la convergence des solveurs, cas *diphasique (2)* sur 50 ans

résultats sont résumés dans les deux tableaux 9 et 10 pour un paramètre fixé à 10 puis 5.

cas	n_{pdt}	$n_{\text{échec Newton}}$	$n_{\text{iter Newton}}$	$n_{\text{iter KSP}}$	$n_{\text{iter KSP utile}}$	$\frac{n_{\text{iter Newton}}}{n_{\text{pdt}}}$
450K	55	11	140	7700	3900	2.54
800K	61	12	168	10600	5800	2.75
1500K	51	9	151	10300	6000	2.96
3M	99	22	262	23300	11500	2.65
6M	67	11	188	18500	10500	2.81

TABLE 9 – Résultats de performances numériques, cas *diphasique (2)* sur 50 ans, `NewtonNiterMax` = 10

Nous obtenons donc de meilleures performances sur ce cas d'application, avec une réduction considérable de la quantité de calcul (ici indiquée par le nombre d'itérations KSP sur l'ensemble de la simulation $n_{\text{iter KSP}}$) d'un facteur 2.3 pour `NewtonNiterMax` = 10 et d'un facteur 3 pour `NewtonNiterMax` = 5. Cela est confirmé par l'examen de la figure 11, qui représente le nombre total d'itérations linéaires pour les différentes valeurs du paramètre `NewtonNiterMax`. On observe une nette réduction entre 40 et 10, puis une relative stabilité entre 10 et 5.

Les temps d'exécution suivent la même amélioration : Lorsque la limite d'itérations maximum est fixée à 10, le temps d'exécution total est amélioré de 57% en moyenne. Lorsqu'elle est fixée à 5, le temps d'exécution est réduit de 67%, soit trois fois plus rapide.

Ces améliorations sont valables pour toutes les tailles de maillage et toutes les configurations testées. De ce fait, la scalabilité du code garde les mêmes proportions.

Les nombres de pas de temps et d'échecs de Newton, pour ce cas, augmentent relativement peu dans les deux scénarios. Ces échecs deviennent beaucoup moins coûteux.

Une question naturelle à la suite de ce travail est de savoir si l'on peut proposer une

cas	n_{pdt}	$n_{\text{échec Newton}}$	$n_{\text{iter Newton}}$	$n_{\text{iter KSP}}$	$n_{\text{iter KSP utile}}$	$\frac{n_{\text{iter Newton}}}{n_{\text{pdt}}}$
450K	63	13	140	5791	3715	2.22
800K	65	13	160	7605	5171	2.46
1500K	67	13	160	8401	5751	2.4
3M	115	28	269	17610	10747	2.34
6M	86	18	183	15157	9385	2.13

TABLE 10 – Résultats de performances numériques, cas *diphasique (2)* sur 50 ans, `NewtonNiterMax` = 5

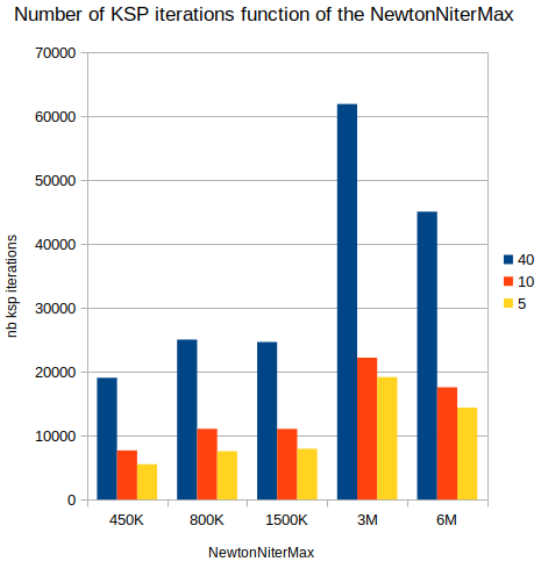

FIGURE 11 – Nombre d'itérations KSP total en fonction de `NewtonNiterMax`, cas *diphasique (2)* sur 50 ans

valeur par défauts pour ces paramètres numériques, les nombres maximum d'itérations linéaires et non-linéaires.

Il convient d'être prudent, et il n'est certain que les résultats précédent puissent être généralisés à tous les cas d'utilisation.

— La valeur limite `NewtonNiterMax` = 5 ne sera pas adaptée au cas *diphasique (1)* par exemple, dont le solveur de Newton converge en moyenne en 7.6 itérations. Par ailleurs une valeur de `NewtonNiterMax` = 8 ralentira l'exécution de ce cas de 10% dans les phases ne produisant pas d'échecs de Newton. Une valeur par défaut entre 8 et 10 nous semble un bon compromis, sachant qu'il est prudent de mener des tests appropriés sur un nouveau modèle, et de contrôler le nombre d'échecs de Newton.

— Par contre, augmenter la valeur de `KSPitmax` au-delà de 150 ne pose pas de problème. La convergence de GMRES peut elle aussi passer par un plateau, mais ne présente en général pas de stagnation.

L'adaptation de ces valeurs à un nouveau modèle devra certainement passer par une phase d'expérimentation.

7 Performances des entrées/sorties

Les écritures de données se font à l'aide de la routine Python de la bibliothèque Numpy `numpy.savez`, dans des fichiers binaires indépendants par rang et par pas de temps d'écriture. Théoriquement, il s'agit d'un moyen performant, qui évite les sur-coûts en temps relatifs à la gestion d'un pointeur d'écriture unique ou dûs à des accès concurrents.

Les performances mesurées, et résumées dans le tableau 11, montrent un comportement qui varie fortement selon la taille du cas étudié, le nombre de rangs et le nombre de pas de temps d'écriture, allant du très significatif au négligeable.

	Métrique	cas monophasique	cas diphasique 3M
E/S	temps E/S (s)	9.6	2
	Volume E/S (MB)	1290	10433
	Appels (nb)	634374	982956
	Débit (MB/s)	134	5277
	Accès E/S individuel (kB)	2.1	10.9

TABLE 11 – Résultats de l'analyse E/S pour les cas : *doublet* (98K inconnues, 30 pas d'écriture), et *diphasique* 3M (1M inconnues, 10 pas d'écriture) sur 48 processeurs

Dans le cas *doublet* que nous avons étudié, une trentaine d'écritures sont effectuées au cours de la simulation, ce qui, sur 48 processeurs, génère plus de 1400 fichiers. Nous observons un temps d'E/S significatif et très variable d'une exécution à l'autre, qui impacte beaucoup le temps de simulation. Ce temps d'écriture varie de 4 à 16 secondes pour un temps d'exécution total de 38 à 50 secondes. L'étude Darshan pour ce cas montre un faible débit des données. Cela s'explique par le grand nombre d'accès aux disques par rapport au petit volume de données. Le coût d'envoi du message devient alors trop grand par rapport au coût de transfert des données contenues dans le message. Ce temps est passé en grande partie dans les accès en écriture, et particulièrement dans les opérations liées aux méta-données des fichiers (figure 12), du fait de leur grand nombre.

La trace des E/S généré par Darshan ainsi que l'instrumentation manuelle montrent un déséquilibre aléatoire des temps d'écriture au niveau des différents rangs, qui va du simple au double, et qui se répercute sur les temps de synchronisation au niveau des communications MPI suivantes.

Le cas *diphasique* étudié est configuré pour faire 5 écritures des données sur 50 années simulées. Cela représente un rapport plus représentatif entre calculs et écriture. Toutes les tailles de maillage montrent un temps E/S petit et raisonnable, qui devient négligeable (<1% du temps total) pour les plus gros cas. la deuxième colonne du tableau 11 pour le cas 3M montre un bien plus gros débit, dû à de plus gros accès. Le déséquilibre entre les processeurs est toujours présent, mais son ampleur est beaucoup moins importante par rapport au coût de calcul total.

Enfin, une routine de post-traitement permet de transformer les fichiers créers par ComPASS dans un format accepté par Paraview (pour l'instant le format natif `vtu` / `pvtu`. Cette opération est séquentielle et peu coûteuse.

Références

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes,

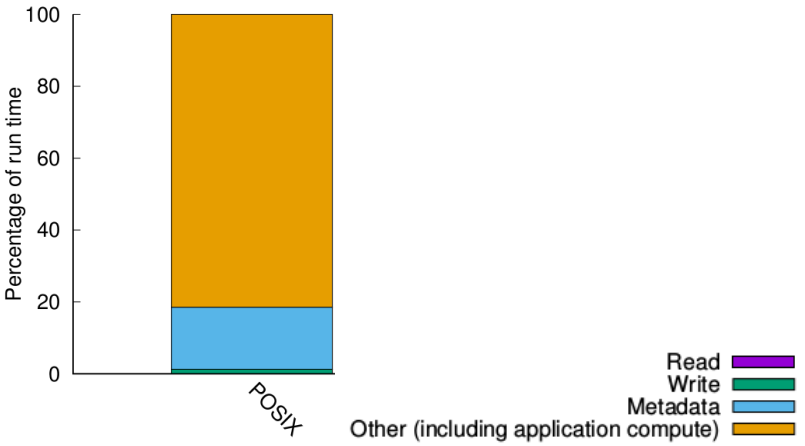

FIGURE 12 – Profil des entrées/sorties en % du temps total, en fonction du type d'accès, généré par Darshan, pour le cas *doublet* sur 48 processeurs

R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.9, Argonne National Laboratory, 2018.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] Dalissier, E., Guichard, C., Havé, P., Masson, R., and Yang, C. Compass : a tool for distributed parallel finite volume discretizations on general unstructured polyhedral meshes. *ESAIM : Proc.*, 43 :147–163, 2013.

[4] R. Eymard, C. Guichard, R. Herbin, and R. Masson. Vertex-centred discretization of multiphase compositional Darcy flows on general meshes. *Computational Geosciences*, 16(4) :987 – 1005, Sept. 2012.

[5] R. Falgout and U. Yang. HYPRE : a library of high performance preconditioners. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Computational Science - ICCS 2002 Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 631–641. Springer-Verlag, 2002. UCRL-JC-146175.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation : Practice and Experience*, 22(6) :702–719, 2010.

[7] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P : A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] S. Lacroix, Y. V. Vassilevski, and M. F. Wheeler. Decoupling preconditioners in the implicit parallel accurate reservoir simulator (IPARS). *Numer. Linear Algebra Appl.*, 8(8) :537–549, 2001. Solution methods for large-scale non-linear problems (Pleasanton, CA, 2000).

[9] S. Lopez, R. Masson, L. Beaude, N. Birgle, K. Brenner, M. Kern, F. Smaï, and F. Xing. Geothermal Modeling in Complex Geological Systems with the ComPASS Code. In *Stanford Geothermal Workshop 2018 - 43rd Workshop on Geothermal Reservoir Engineering*, Stanford, United States, Feb. 2018. Stanford University.

[10] R. Masson, P. Quandalle, S. Requena, and R. Scheichl. Parallel preconditioning for sedimentary basin simulations. In *International Conference on Large-Scale Scientific Computing*, pages 93–102. Springer, 2003.

[11] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2 edition, 2003.

[12] R. Scheichl, R. Masson, and J. Wendebourg. Decoupling and block preconditioning for sedimentary basin simulations. *Comput. Geosci.*, 7(4) :295–318, 2003.

[13] F. Xing, R. Masson, and S. Lopez. Parallel numerical modeling of hybrid-dimensional compositional non-isothermal Darcy flows in fractured porous media. *Journal of Computational Physics*, 345 :637–664, May 2017.